OCaml 시작하기

2012.11.13
2012 가을, 프로그래밍의 원리
강동옥,최민아
{dokang,machoi}@ropas.snu.ac.kr
서울대학교 프로그래밍연구실

OCaml을 배워봅시다

컴파일러 설치

- snucse 계정이 있으면 martini 등 snucse 서버 사용
  - martini에는 3.10.2 버전
  - 최신버전은 3.12.1 (금지)
  - 연구실에선 3.11.2 사용
- ocaml.org
  - caml.inria.fr
- Download → 각자 OS에 맞는 binary 받기
- 혹은 apt-get(Ubuntu), port(MAC) 패키지 매니저

편집기

- vi, emacs
  - 종교
  - 편하신대로
- eclipse 플러그인(OcaIDE)+Cygwin
  - 설치방법 http://ropas.snu.ac.kr/~ta/4190.210/12/practice/ocaml_tutorial.pdf
- 메모장(Notepad)…?
- 기타
  - 문법 강조가 되는 것

실행기, 소스파일

- 실행기
  - 식을 입력하면 바로 결과를 볼 수 있습니다.
  - 식을 입력하고 ;;로 맺습니다.
  - 끌 땐 #quit;;
  - ml 파일을 오픈하고 싶으면 ocaml -init filename.ml

- 소스 파일
  - *.ml 파일을 작성해 컴파일합니다.
  - ;;를 붙이지 않습니다.
  - ml 파일은 정의(let) 들의 집합입니다.
  - 과제로는 컴파일이 되는 .ml 파일을 제출합니다.

새로운 실습언어라니..

- Scheme 이제야 겨우 익숙해졌는데..
- OCaml은 또 어디서 굴러온 녀석이지?

스킴하고 비교해볼게요

스킴 vs. 오켐엘

- 스킴에서 보았던 수많은 괄호들..

Untitled - DrScheme*

File Edit View Language Scheme Special Help

Untitled (define ...) Save Debug Macro Stepper Check Syntax Run Stop

```
  (cond ((not (is-sol? sol)) (error "Error : not solution"))
        ((isS? sol) (display "S"))
        ((isK? sol) (display "K"))
        ((isI? sol) (display "I"))
        ((isv? sol) (begin (display "\"") (display (var sol)) (display "\"")))
        ((isa? sol) (begin (display "(") (pprint (al sol)) (display " ") (pprint (ar sol)) (display
")")))
        )
  )

(define (react sol)
  (define (reaction sol)
    (cond ((or (isS? sol) (isK? sol) (isI? sol) (isv? sol)) sol)
          ((isI? (al sol)) (reaction (ar sol)))
          ((and (isa? (al sol)) (isK? (al (al sol)))) (reaction (ar (al sol))))
          ((and (isa? (al sol)) (isa? (al (al sol))) (isS? (al (al (al sol)))))
           (reaction (a (a (ar (al (al sol))) (ar sol)) (a (ar (al sol)) (ar sol)))))
          (else (cond ((equal? (a (reaction (al sol)) (reaction (ar sol))) sol) sol)
                      (else (reaction (a (reaction (al sol)) (reaction (ar sol))))))
                )
          )
    )
  (pprint (reaction sol))
```

Welcome to DrScheme, version 372 [3m].
Language: Pretty Big (includes MrEd and Advanced Student).

스킴 vs. 오캠엘

- 이젠 안녕

```
 1 let rec length l = match l with
 2   | [] -> 0
 3   | _::t -> 1 + length t
 4
 5 (* tail-recursive version of length *)
 6 let length' l =
 7   let rec f l result = match l with
 8     | [] -> result
 9   | _::t -> f t (result+1)
10   in
11   f l 0
12
13 let hd l = match l with
14   | [] -> raise (Failure "hd")
15   | h::_ -> h
16
17 let tl l = match l with
18   | [] -> raise (Failure "tl")
19   | _::t -> t
```

스킴 vs. 오켐엘

- Prefix 연산자
- (- (* 4 5) 2)

- Infix 연산자
- 4 * 5 - 2

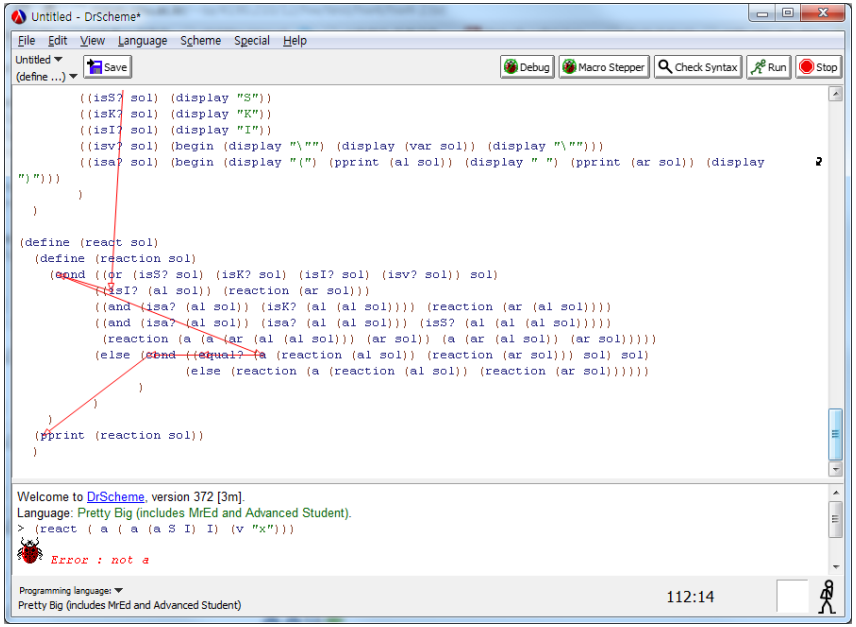
스킴 vs. 오켐엘

- If문에서 false와 true에 대해 자유롭게 다른 타입 값 리턴 가능
- (if b 3 false)

- 타입이 일치 해야만함.
- If b 3 4
- If c true false
- If d “a” “b”

정말 울어야 할일?

- 스킴 코딩하다보면 자주 볼 수 있는 빨간줄..
- 오캠엘은 강한 타입시스템으로 안전한 프로그래밍을 돕는다.



값 만들기 / 타입추론

- 정수
  - `let i = 1` `(* int *)`
- 실수
  - `let f = 3.2` `(* float *)`
- 문자열
  - `let s = "hello world!"` `(* string *)`
- 리스트
  - `let l = [1;2;3;4;5]` `(* int list *)`
  - `let l2 = 1::2::3::4::5::[]` `(* int list *)`
- 페어
  - `let p = (3, 5)` `(* int * int *)`
- 함수
  - `let incr = fun x -> x+1` `(* int -> int *)`
  - `let cons a b = a::b`
    `(* 'a -> 'a list -> 'a list *)`

값 만들기 / 타입추론

- 직접 정의한 타입도 유추해 줍니다.

```
type tree = Leaf of int
          | Node of tree list

let t1 = Leaf 3           (* tree *)
let t2 = Leaf 5           (* tree *)
let t3 = Node [t1;t2]     (* tree *)
```

값 만들기 / 타입추론

- 여러 가지 조합된 값도 타입을 유추해줍니다.

```
type tree = Leaf of int
          | Node of tree list

let a = [(3, Node [Leaf 3; Leaf 2])]
                    (*        ???       *)
```

값 만들기 / 타입추론

- 여러 가지 조합된 값도 타입을 유추해줍니다.

```
type tree = Leaf of int
          | Node of tree list

let a = [(3, Node [Leaf 3; Leaf 2])]
                    (* (int * tree) list *)
```

값 만들기 / 타입추론

- 그런데 만든 값을 어떻게 사용하지?

```
type tree = Leaf of int
          | Node of tree list

let a = [(3, Node [Leaf 3; Leaf 2])]
                    (* (int * tree) list *)
```

IF문

- 간단한 값 비교는 IF문으로

```
let abs n =
   if n < 0 then -n else n

let sign n =
   if n = 0 then 0
   else if n < 0 then -1
   else 1
```

IF문

- If문을 이용해 gcd를 짜보자

```
let rec gcd a b =     (* 재귀함수는 꼭 rec를 붙여요! *)
   if a=1 || b=1 then 1
   else if a=b then a
   else if a<b then gcd b a
   else gcd (a-b) b
```

IF문

- 간단한 기본타입비교에 훌륭함
- 약간 복잡한 값 비교도 해냄
  - tree라는 값은 Leaf 3일까? Leaf 2일까?

```
let what_is tree =
   if tree = Leaf 3 then 1
   else if tree = Leaf 2 then 2
   else 3
```

- 좀더 복잡한 비교는 한계가 있음
  - my_list는 헤드 원소가 1인 리스트일까?

패턴 매칭

- match - with
- 함수 외에 어떤 타입 값이든 매칭 가능!
  - match x with
    A -> a
    | B -> b
    | C -> c
    | _ -> default

패턴 매칭

- 기본 타입 값 매칭

```
let what_is_x x =
  match x with 1 -> “one” | _ -> “else”

let one_or_two p =
  match p with
  | (1, “apple”) -> true
  | (2, “orange”) -> true
  | _ -> false

let rec length l =
  match l with
  | hd :: tl -> 1 + (length tl)
  | [] -> 0
```

패턴 매칭

- 직접 만든 타입 값 매칭

```
let is_Leaf tree =
   match tree with
   | Leaf v -> true
   | Node tree_list -> false
```

패턴 매칭

- 여러 가지 조합된 값도 패턴매칭이 됩니다.

```
let my_list = [(3, Node [Leaf 3; Leaf 2])]

let rec get_n_leaf ml n =
  match ml with
  | (m, Leaf v) :: tl ->
      if m = n
      then Leaf v
      else get_n_leaf tl n
  | (_, _) :: tl -> get_n_leaf tl n
  | [] -> raise (…)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠 구멍에 끼워야

```
let one = 1                              (* int *)

let two_dot_two = 2.2                 (* float *)

let add = (+)  (* int -> int -> int *)

let three_dot_two =
    add one two_dot_two  (* error *)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠 구멍에 끼워야

```
let one = 1                              (* int *)

let two_point_two = 2.2                  (* float *)

let add = (+)  (* int -> int -> int *)

let three_dot_two =
    (float_of_int one) +.
    two_dot_two                          (* float *)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠 구멍에 끼워야

```
let first p =                (* int * string -> bool *)
  match p with
  | (1, “apple”) -> true
  | (2, “orange”) -> true
  | _ -> false

let first_of_p = first (“1”, “one”)   (* error *)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠 구멍에 끼워야

```
let first p =                (* int * string -> bool *)
  match p with
  | (1, “apple”) -> true
  | (2, “orange”) -> true
  | _ -> false

let first_of_p = first (1, “one”)  (* bool *)
```

강한 타입 체계

- OCaml은 똑똑합니다.
- get_n_leaf 함수의 인자 ml에 my_list같이 생긴 값이 들어온다고 알려주지 않지만 OCaml은 알아냅니다.

```
let my_list = [(3, Node [Leaf 3; Leaf 2])]

let rec get_n_leaf ml n =
  match ml with
  | (m, Leaf v) :: tl ->
      if m = n then Leaf v
      else get_n_leaf tl n
  | (_, _) :: tl -> get_n_leaf tl n
  | [] -> raise (…)
(*get_n_leaf : ('a * tree) list -> 'a -> tree *)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠구멍에 넣어야

```
let bad_list = [(3, "this is string")]

let rec get_n_leaf ml n =
  match ml with
  | (m, Leaf v) :: tl ->
      if m = n then Leaf v
      else get_n_leaf tl n
  | (_, _) :: tl -> get_n_leaf tl n
  | [] -> raise (…)
(*get_n_leaf : ('a * tree) list -> 'a -> tree *)
let leaf = get_n_leaf bad_list   (* error *)
```

강한 타입 체계

- 열쇠는 꼭 맞는 열쇠구멍에 넣어야

```
let my_list = [(3, Node [Leaf 3; Leaf 2])]

let rec get_n_leaf ml n =
  match ml with
  | (m, Leaf v) :: tl ->
      if m = n then Leaf v
      else get_n_leaf tl n
  | (_, _) :: tl -> get_n_leaf tl n
  | [] -> raise (…)
(*get_n_leaf : ('a * tree) list -> 'a -> tree *)
let leaf = get_n_leaf my_list   (* tree *)
```

마음의 고향 : 나무

```
(define (leaf t) (cons 'leaf t))

(define (node tree_list) (cons 'node
tree_list))

(define (is-leaf? tree) (equal? 'leaf
(car tree)))

(define (leaf-val tree) (cdr tree))

(define (nth-child tree nat)
  (define (nth-elem tree_list nat)
    (cond ((= nat 0) (car tree_list))
          (else (
             nth-elem
             (cdr tree_list) (- nat 1)))
          )
    )
  (if (is-leaf? tree)
      (error "tree is leaf")
      (nth-elem (cdr tree) nat))
  )
```

- 스킴에서 많이 다뤘던 나무구조
- OCaml에서 나무를 만들고/사용하는 함수를 정의해보겠습니다

나무 타입 정의

- OCaml은 타입을 직접 정의할 수 있습니다.
- type constructor를 붙여서 정의합니다.
- 언제나 대문자로 시작!

```
type tree = Leaf of int
          | Node of tree list
```

나무 타입 정의

- 좀더 일반적으로
- Type function

```
type 'a tree = Leaf of 'a
             | Node of 'a tree list
```

나무를 만드는 함수

- type constructor를 이용해 나무를 만들 수 있습니다.

```
let leaf v = Leaf v

let node tree_list = Node tree_list
```

나무를 사용하는 함수

- match를 이용해서 나무를 사용할 수 있습니다.

```
let is_leaf tree =
   match tree with
   | Leaf v -> true
   | Node _ -> false

let leaf_val tree =
   match tree with
   | Leaf v -> v
   | node _ -> raise (…)
```

나무를 사용하는 함수

- match를 이용해서 나무를 사용할 수 있습니다.

```
let nth_child tree n =
   let rec nth_elem l n =
      match l with
      | hd :: tl ->
         if n=0 then hd
         else nth_elem tl (n-1)
      | [] -> raise (…)
   in
   match tree with
   | Leaf v -> raise (…)
   | Node tree_list -> nth_elem tree_list n
```

나무를 사용하는 함수

- 조금 복잡한데?

```
let nth_child tree n =
   let rec nth_elem l n =
      match l with
      | hd :: tl ->
         if n=0 then hd
         else nth_elem tl (n-1)
      | [] -> raise (…)
   in
   match tree with
   | Leaf v -> raise (…)
   | Node tree_list -> nth_elem tree_list n
```

나무를 사용하는 함수

- match로 한방에! 더 간단하게!

```
let rec nth_child tree n =
   match (tree, n) with
   | (Node(hd::tl), 0) -> hd
   | (Node(hd::tl), _) ->
        nth_child (Node tl) (n-1)
   | _ -> raise (…)
```

예외처리

- 에러를 내야 하는 케이스 어떻게 할까?
- 기본 제공 예외처리 VS 예외처리 직접 만들기
- raise 사용

```
let f x =
  raise Not_Found          (* 인자가 없는 예외 *)

let f x =
  raise (Invalid_argument "error")
                           (* 인자를 가진 예외 *)
```

예외처리

- 예외를 정의해보자.

```
exception Not_leaf
exception Error of string
```

- 사용할땐

```
raise Not_leaf
raise (Error “this is leaf”)
```

OCaml 버전 완성

- 만들기 : type constructor
- 사용하기 : match-with

```
type 'a tree = Leaf of 'a
             | Node of 'a tree list
let rec nth_child tree n =
   match (tree, n) with
   | (Node(hd::tl), 0) -> hd
   | (Node(hd::tl), _) ->
        nth_child (Node tl) (n-1)
   | _ -> raise Not_found
```

OCaml 버전 완성

```
(define (leaf t) (cons 'leaf t))

(define (node tree_list)
   (cons 'node tree_list))

(define (is-leaf? tree)
   (equal? 'leaf (car tree)))

(define (leaf-val tree) (cdr tree))

(define (nth-child tree nat)
  (define (nth-elem tree_list nat)
    (cond ((= nat 0) (car tree_list))
          (else
              (nth-elem
                (cdr tree_list) (- nat 1)))
          )
    )
  (if (is-leaf? tree)
      (error "tree is leaf")
      (nth-elem (cdr tree) nat))
  )
```

```
type 'a tree = Leaf of 'a
             | Node of 'a tree list
let rec nth_child tree n =
   match (tree, n) with
   | (Node(hd::tl), 0) -> hd
   | (Node(hd::tl), _) ->
        nth_child (Node tl) (n-1)
   | _ -> raise Not_found
```

구지 함수로 인터페이스

```
(define (leaf t) (cons 'leaf t))

(define (node tree_list)
   (cons 'node tree_list))

(define (is-leaf? tree)
   (equal? 'leaf (car tree)))

(define (leaf-val tree) (cdr tree))

(define (nth-child tree nat)
  (define (nth-elem tree_list nat)
    (cond ((= nat 0) (car tree_list))
          (else
              (nth-elem
                (cdr tree_list) (- nat 1)))
          )
    )
  (if (is-leaf? tree)
      (error "tree is leaf")
      (nth-elem (cdr tree) nat))
  )
```

```
type 'a tree = Leaf of 'a
             | Node of 'a tree list
let leaf t = Leaf t
let node tree_list = Node tree_list
let is_leaf tree =
     match tree with
     |  Leaf v -> true
     |  Node tree_list -> false
let leaf_val tree =
     match tree with
     |  Leaf v -> v
     |  Node tree_list -> raise Not_Found
let rec nth_child tree n =
   match (tree, n) with
   | (Node(hd::tl), 0) -> hd
   | (Node(hd::tl), _) ->
        nth_child (Node tl) (n-1)
   | _ -> raise Not_found
```

조금 다른 스타일로

- 인자를 페어로 받을 수 있습니다.

```
let rec nth_child (tree, n) ->
     match (tree, n) with
     | (Node(hd::tl), 0) -> hd
     | (Node(hd::tl), _) ->
          nth_child ((Node tl), (n-1))
     | _ -> raise (…)
```

조금 다른 스타일로

- 익명함수를 이용할 수도 있습니다

```
let nth_child =
  fun tree n ->
     match (tree, n) with
     | (Node(hd::tl), 0) -> hd
     | (Node(hd::tl), _) ->
          nth_child (Node tl) (n-1)
     | _ -> raise (…)
```

조금 다른 스타일로

- 페어인자 + 익명함수 + match를 한방에

```
let nth_child =
  function
       (Node(hd::tl), 0) -> hd
     | (Node(hd::tl), _) ->
          nth_child ((Node tl), (n-1))
     | _ -> raise (…)
```

커리 함수

- 잠깐! 함수를
- func x1 x2 x3 로도 쓸 수 있고
- func (x1, x2, x3) 로도 쓸 수 있다고?
- 무슨차이지?

커리 함수

- func x1 x2 x3 는 인자를 하나만 받을수도..
- func (x1, x2, x3) 는 인자를 한꺼번에 받아야..
- 따라서 curry함수는 이런것도 가능

```
let add a b = a + b

let incr x = add 1
```

커리 함수

- 과제에서 타입이 틀리면 0점입니다
  - int -> int -> int
    - let add a b = …
  - (int * int) -> int
    - let add (a,b) = …
  - 주의하세요
  - 제발
  - 이번 튜토리얼의 핵심!

코드에 타입 명시하기

- 정수
  - `let i : int = 1 (* int *)`
- 문자열
  - `let s : string = “hello world!” (* string *)`
- 리스트
  - `let l : int list = [1;2;3;4;5] (* int list *)`
  - `let l2 : int list = 1::2::3::4::5::[] (* int list *)`
- 함수
  - `let incr : int -> int = fun x -> x+1 (*int -> int *)`
  - `let cons : ‘a -> ‘a list = fun a b -> a::b`
    `(* ’a -> ’a list -> ’a list *)`
- 기타 등등

Print 함수들

- Print 함수들은 리턴타입이 unit입니다.

- print_char : char -> unit
- print_string : string -> unit
- print_int : int -> unit
- print_float : float -> unit
- print_endline : string -> unit

세미콜론

- 스킴에서와 동일하게 세미콜론(;)을 이용하여 순서대로 실행할 수 있습니다.
- C에서는 statement의 끝을 의미하지만 OCaml에서는 두 식이 순차적으로 실행된다는 의미입니다.
- 세미콜론은 오른쪽 값을 리턴합니다.

```
let my_print nat str =  (*int -> string -> int*)
    print_int nat ;
    print_string “, ”^str;
    nat
```

물건중심 문법

- 참조타입(reference)

```
let a = ref 1                (* int ref *)

let b = !a                 (* b = 1, int *)

let c =
  a := 2 ; !a              (* c = 2, int *)
```

물건중심 문법

- 배열(array)

```
let a = [| 1; 2; 3 |]                (* int array *)

let b = a.(0)                        (* b = 1, int *)

let c = a.(0) <-4 ; a.(0)            (* c = 4, int *)
```

try-with:예외잡기

- Java의 try-catch와 비슷

```
let s =  (*성공하면 "a/b결과", 실패하면 "error" *)
  try
     string_of_int (
        let a = read_int() in
        let b = read_int() in
        a/b)
  with Division_by_zero -> "error"
```

try-with:예외잡기

- Java의 try-catch와 비슷

```
let s =  (*성공하면 “a/b결과”, 실패하면 “error” *)
  try
     string_of_int (
        let a = read_int() in
        let b = read_int() in
        a/b)
  with _ -> “error”
```

표준 라이브러리

- OCaml 사용자 매뉴얼
  - http://caml.inria.fr/pub/docs/manual-ocaml
- 라이브러리 사용법
  - http://caml.inria.fr/pub/docs/manual-ocaml/libref/index_modules.html
  - Pervasives, List, String 등을 참조
- 우리의 영원한 친구 구글

표준 라이브러리

- List 모듈을 사용해 볼까요

```
let a = [1;2;3]                (*a=[1;2;3]*)

let b = List.tl a               (*b=[2;3]*)

let c = List.length a             (*c=3*)
```

표준 라이브러리

- Pervasive에 들어있는 함수는 그냥 써도 돼요!

```
let a = 1                          (*a=1*)

let b = string_of_int a            (*b="1"*)
```

지난학기 튜토리얼들

- 2011년 봄학기 튜토리얼 (by 이원찬)
  - http://ropas.snu.ac.kr/~ta/4190.310/11s/ocaml_tutorial.pdf
- 2011년 가을학기 튜토리얼 (by 윤용호,김진영)
  - http://ropas.snu.ac.kr/~ta/4190.310/11/ocaml_tutorial11f.pdf

하나씩 걸어두세요

- OCaml 치트시트
  - http://www.lri.fr/~conchon/docs/ocaml-lang.pdf
- 웹에서 OCaml 인터프리터 사용 및 튜토리얼
  - http://try.ocamlpro.com/